תרגיל בית 4

234124 - מבוא לתכנות מערכות

שם: פרח קב

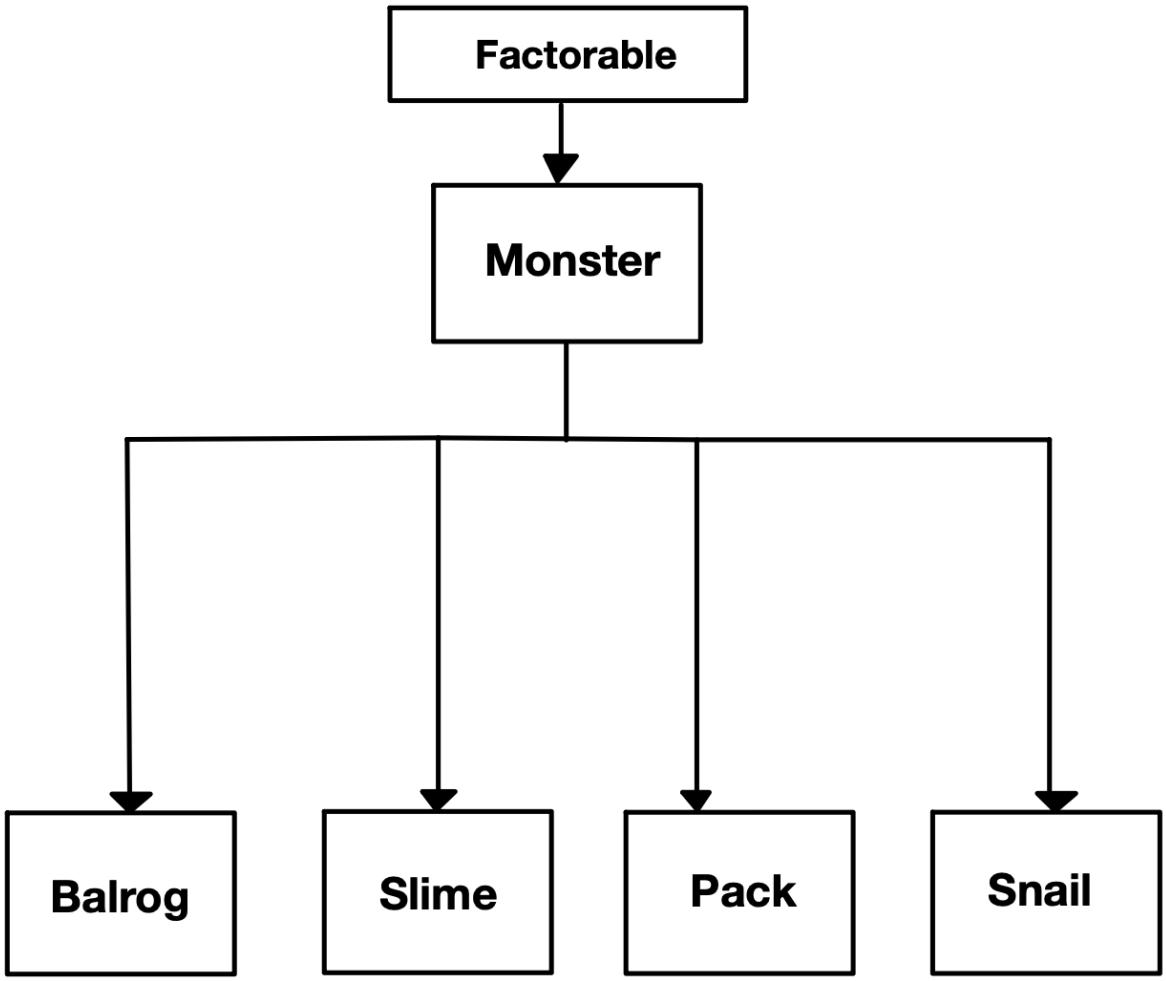
ת״ז: 213309388

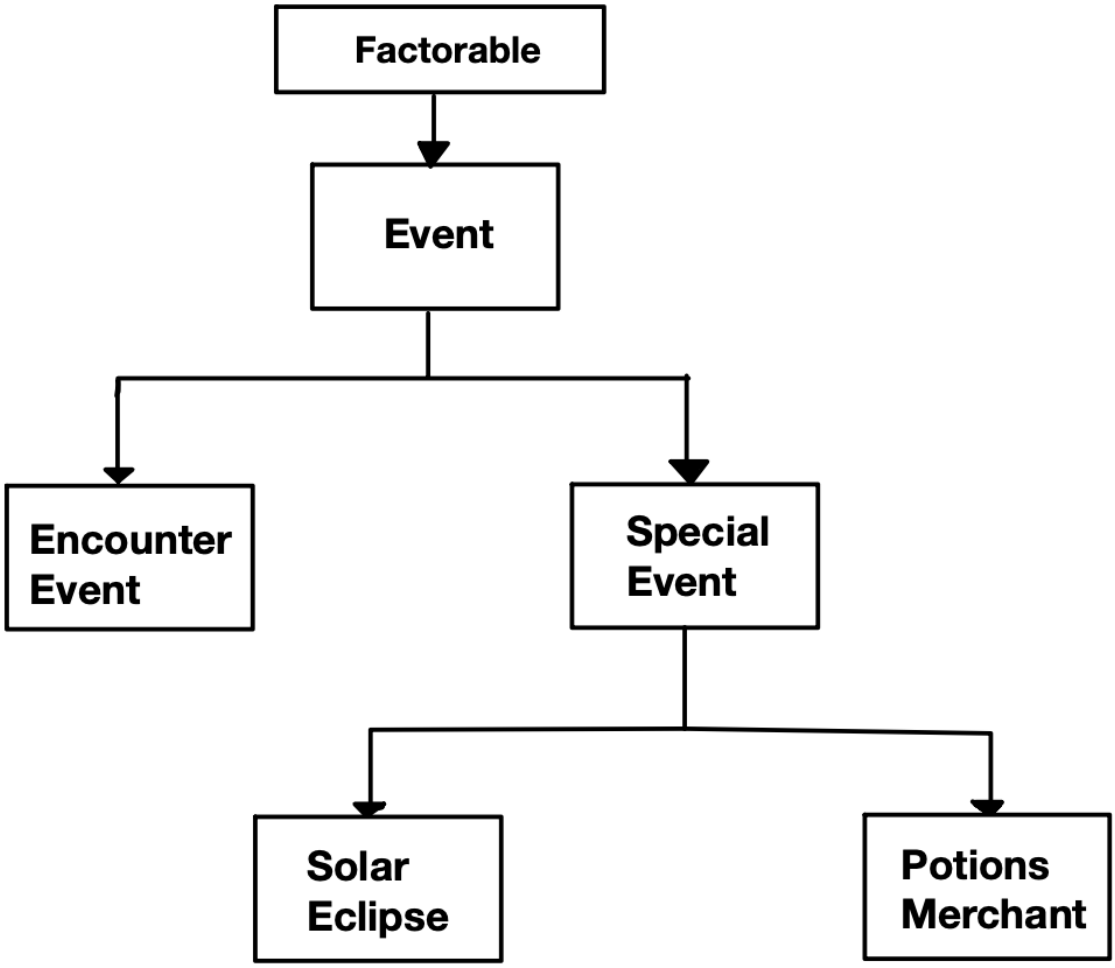
שם: ג׳מאל אבו מוך

ת״ז: 213754088

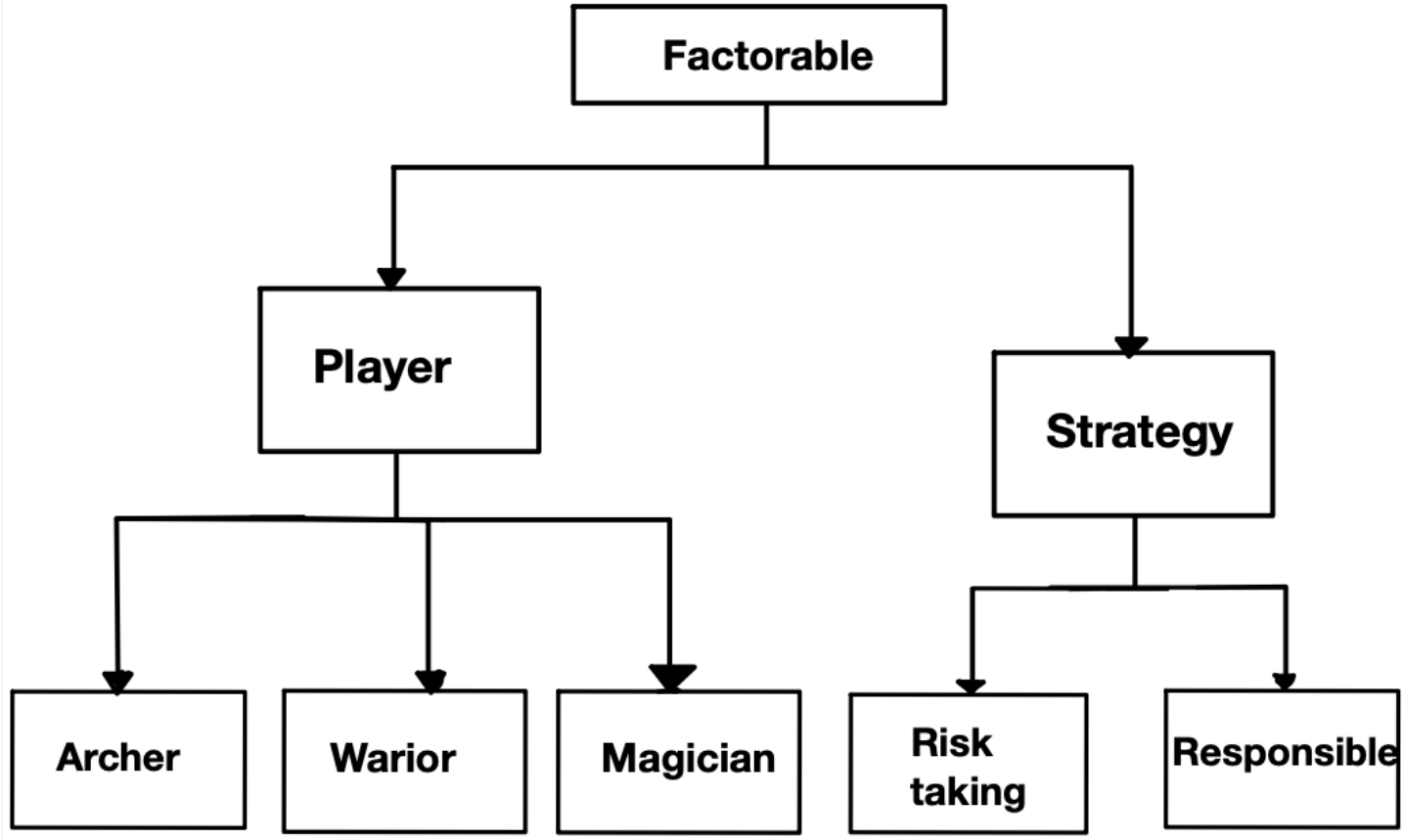
חלק יבש 4

1. מחלקות UML הן:





Potion Merchant Offer



MatamStory

Type Info Does Not Exist

2. ה design patterns שהתמשנו בהן הן:

Factory pattern:

עבור המחלקה Factorable<Type> השתמשנו בfactory.

Every class that inherits from `Factorable<Type>` gets a static method ‘registerFactory’ that takes a key for the sub-type, and lambda, which in turn saves them in a static singleton `std::map` for each type template, which defines how the type should be constructed.
To create an instance the caller calls “Type”::createType that takes the requested sub-type key, vector arguments with the type string,
The function returns a shared_ptr<T> which is an instance created by the appropriate lambda in the factories map.
This pattern enables us to dynamically add types to the system, and offers the possibility for extending the system without having to worry about changing a lot of places, and avoids having other parts of the system knowing about subtypes in compile-time, and provides a clean way of dividing responsibility, in which the caller does not have to know how to construct the requested type (where this task is passed to the type itself),
Bonus: provides a clean way to use a nested inheritance design.

Strategy pattern:

Used in `Strategy` class which is inherited by `RiskTaking` and `Responsible` that defines how a player with the said character behaves,

This pattern allows us to have a delegate way of having different behaviors across players with different jobs, it also creates the possibility of changing the character mid-game.

Polymorphism:

To have a unified call for all the events, an`Event` interface is introduced, which is inherited by `EncounterEvent` and `SpecialEvent` that override the function to provide an implementation according to their need.

The same with the `Player` class, where different jobs do not have a core difference between them, hence just overriding the parts in which they differ.

The same goes for `Monster` Type, where they all behave like a `record` of maintaining the stats about their power, And allowing the special type `Pack` of acting like a monster, but with overridden methods

3. In our implementation, it is not out-of-the-box feature, First, we have to change the `Monster` class to inherit from `EncounterEvent` instead of factorable, then add a `Rouge` class with the proper handling of registering itself in the factory (using `registerFactory`), then, in the `Rouge` class, we must override the `applyTo` method, in which the encounter behavior is changed with adding the requested check and deciding how to continue, if proper, the `Rouge` class could use the `base::appleTo`, after performing it's check.
   To divide responsibilities and avoid code duplication, more methods can be introduced on `EncounterEvent` describing the `postEncounter `and `preEncounter` behaviors.
   Another implementation design: a `Monster Strategy` can be introduced using the strategy pattern, which could have its own custom handling of every encounter.
   Bonus (not preferred): a method can be introduced on `Monster` class that reports back to the `EncounterEvent` whether to continue the fight or withdraw.

4. Technically, it would be possible with cast, because the player's jobs are in fact inheritance of the class `Player`, the problem is that there are undefined behaviors, e.g.:
   The `maxHp` for the `Warrior` is 150, assume his current life is 120, but the target class has `maxHp` 100, do we change his current life to 100 or do we change it according to the same percent as it was with the previous `maxHp`, that is the reason we need to explicitly define the casting by defining this behavior in each inherited class and describe how its casted to a different player job.
   And to avoid hard-writing the requested class type, we must extend the system with providing a way to cast each to type to another without hard-checking the type in compile-time.
   For example, a static method called `cloneFrom` that implements that factory pattern can be introduced to `Player` in which each subtype registers a `cloner`, that takes a pointer to a player class, and returns `static_cast<SubType*>(value)`, this method is called on the base type `Player` and handles the mapping accordingly.
   This way we can avoid having to know about the `Player` job subtypes in the `EncounterEvent` at compile-time.